À la pêche aux groupes avec Python

PAR RICHARD GOMEZ

juillet 2017

À l'occasion de la sortie de son aide-mémoire pour Python 3 aux éditions Ellipses (Le petit Python*, mai 2017), Richard Gomez nous propose une récréation amusante sur les groupes. Il s'agit d'écrire un programme capable de générer tous les groupes finis à isomorphisme près. Cet article a été publié dans la revue Mathématice et sur le site Mégamaths.*

Résumé

La structure d'un groupe est contenue dans sa table de Pythagore. Si on numérote les éléments d'un groupe fini d'ordre n, cette table s'identifie naturellement à une matrice carrée d'ordre n à coefficients dans $\{1, ..., n\}$. Il est clair que dans une telle matrice, chaque ligne et chaque colonne est une permutation de $(1, ..., n)$. Dans le présent article, une telle matrice est appelée *sudoku*. Il est facile d'écrire un programme en Python générant tous les sudokus d'un ordre donné, mais attention : un sudoku quelconque ne définit pas forcément un groupe. En revanche, si la loi interne définie par un sudoku est associative, alors on a affaire à un groupe. Tester une matrice pour savoir si elle est « associative » n'est pas difficile. Nous proposons ici un programme capable de trouver tous les sudokus associatifs d'ordre inférieur ou égal à 6 (au delà, les calculs prennent trop de temps). Notre programme fait ensuite un tri : il ne garde qu'un groupe par classe d'isomorphisme. Au final, on se retrouve avec la liste complète des groupes d'ordre au plus 6, à isomorphismes près.

1 Une caractérisation des lois de groupe

On rappelle les notions de *loi interne* (loi de composition interne) et de *groupe*.

Définition 1. *Soit E un ensemble. Une loi de composition interne dans E, est une application de $E \times E$ vers E. L'image de (a, b) est noté $a \times b$ ou encore $a\, b$ (notation multiplicative).*

Définition 2. *Un groupe est un couple $(G, \times)$ où G est un ensemble non vide et $\times$ une loi de composition interne dans G vérifiant les trois propriétés ci-dessous.*

- *Pour tous $a, b, c \in G$, $(a\, b)\, c = a\, (b\, c)$.*
- *Il existe $e \in G$ tel que pour tout $a \in G$, $a\, e = e\, a = a$.*
- *Pour tout $a \in G$, il existe $a' \in G$ tel que $a\, a' = a'\, a = e$.*

Une loi vérifiant la première propriété est dite *associative*. L'élément e exhibé dans la deuxième propriété s'appelle *élément neutre* de G. L'élément a' de la troisième propriété est appelé *symétrique* de a. On invite le lecteur à voir ou revoir les propriétés fondamentales des groupes dans [1] ou [4], par exemple. Nous donnerons dans cette section une condition nécessaire et suffisante pour qu'une loi interne soit une loi de groupe.

Définition 3. *Une structure associative est un couple $(E, \times)$ où E est un ensemble non vide et $\times$ une loi de composition associative de E.*

Une structure associative n'est pas un groupe : l'associativité est nécessaire, mais loin d'être suffisante.

Exercice 1. Donner des exemples de structures associatives qui ne sont pas des groupes.

Définition 4. *Une structure sudoku est un couple $(E, \times)$ où E est un ensemble non vide et $\times$ une loi interne dans E vérifiant les deux propriétés suivantes.*

- *Pour tous $a, b \in E$, il existe un unique $x \in E$ tel que $a\, x = b$.*
- *Pour tous $a, b \in E$, il existe un unique $x \in E$ tel que $x\, a = b$.*

On notera que dans une structure sudoku, chaque élément est *régulier à gauche*, autrement dit, une égalité $a\,x = a\,y$ implique l'égalité $x = y$ (le lecteur le vérifiera par lui-même). De même, chaque élément est *régulier à droite*, ce qui signifie qu'une égalité $x\,a = y\,a$ implique $x = y$. La propriété qui suit affirme que la notion de groupe équivaut à la notion de structure sudoku associative.

Proposition 5. *Soit E un ensemble muni d'une loi interne $\times$. Alors $(E, \times)$ est un groupe si et seulement si $(E, \times)$ est une structure sudoku associative.*

Démonstration. Tout groupe est clairement une structure sudoku associative (on invite néanmoins le lecteur novice à le vérifier). Il s'agit donc d'établir la réciproque. Soit $(E, \times)$ une structure sudoku associative.

1. Soit $a \in E$. Soit $e \in E$ tel que
$$a\,e = a$$
Un tel élément existe puisque E est une structure sudoku. Montrons que e est neutre à gauche. Soit $x \in E$. De $a\,e = a$ on déduit $a\,e\,x = a\,x$, c'est-à-dire $a\,(e\,x) = a\,x$, d'où, par régularité, $e\,x = x$.
2. Soit $f \in E$ tel que $f\,a = a$. Montrons que f est neutre à droite. Soit $x \in E$. De $f\,a = a$, on déduit $x\,f\,a = x\,a$, c'est-à-dire $(x\,f)\,a = x\,a$, d'où, par régularité, $x\,f = x$.
3. Montrons que $e = f$. Puisque f est neutre à droite, $e\,f = e$. Puisque e est neutre à gauche, $e\,f = f$. On a donc $e = f$. Nous avons prouvé que E possède un élément neutre noté e.
4. Montrons que tout élément possède un symétrique. Soit $a \in E$. Il existe alors $a' \in E$ tel que $a\,a' = e$ (nous sommes dans une structure sudoku). Montrons que a' est aussi symétrique à gauche de a. De $a'\,e = a'$ et $e = a\,a'$, on déduit $a'\,a\,a' = e\,a'$, c'est-à-dire $(a'\,a)\,a' = e\,a'$, d'où $a'\,a = e$.
5. La loi $\times$ est supposée associative, et nous avons montré qu'elle vérifie les deuxième et troisième axiomes de la définition d'un groupe. Il s'ensuit que $(E, \times)$ est un groupe. □

Exercice 2. Soit $(E, \times)$ une structure associative. Montrer que $(E, \times)$ est un groupe si et seulement si il vérifie l'*axiome du sandwich* : pour tous $a, b, c \in E$, il existe un et un seul $x \in E$ tel que $a\,x\,b = c$. Pour en savoir plus sur cet axiome, on consultera [7].

2 Sudokus associatifs

Soit E un ensemble fini de cardinal $n > 0$ muni d'une loi $\times$ interne. On numérote les éléments de E de sorte que ce dernier s'écrive
$$E = \{a_1, ..., a_n\}$$
On peut maintenant associer de manière naturelle une matrice P d'ordre n au couple $(E, \times)$. En effet, soit $(i, j) \in \{1, ..., n\}^2$. On note k l'élément de $\{1, ..., n\}$ tel que
$$a_i\,a_j = a_k$$
On pose alors
$$P_{ij} = k$$
La matrice P ainsi définie est à coefficients dans $\{1, ..., n\}$. Bien entendu, P dépend de la manière dont on a numéroté les éléments de E. Néanmoins, une fois numérotés les éléments de E, il y a correspondance bi-univoque entre les lois internes de E, et les matrices d'ordre n à coefficients dans $\{1, ..., n\}$.

Supposons maintenant que $(E, \times)$ est une structure sudoku avec éléments numérotés. On note P la matrice associée. Le fait que chaque équation $a\,x = b$ d'inconnue x possède une et seule solution dans E, implique que chaque ligne de P est une permutation de $(1, ..., n)$. De même, le fait que chaque équation $x\,a = b$ possède exactement une solution, implique que chaque colonne de P est une permutation de $(1, ..., n)$. Dans une telle matrice, aucune ligne ne se répète (aucune colonne non plus). Nous allons nous intéresser à ce type de matrices (pour simplifier notre programme Python à venir, nous ne nous intéresserons qu'au cas où la première colonne et la première ligne correspondent à la permutation triviale).

Définition 6. *Soit P une matrice carrée d'ordre n à coefficients dans $\{1,...,n\}$. On dit que P est un sudoku si chaque ligne et chaque colonne est une permutation de $(1,...,n)$, la première ligne et première colonne étant égales à* $\begin{pmatrix} 1 & ... & n \end{pmatrix}$.

On aura aussi besoin de la définition ci-dessous :

Définition 7. *Soit P une matrice d'ordre n à coefficients dans $\{1, ..., n\}$. On dit que P est associative si la loi définie par P est associative.*

La proposition ci-dessous est une simple conséquence de la proposition 5 de la section précédente :

Proposition 8. *Soit G un groupe fini. On numérote les éléments de G de sorte que le neutre porte le numéro 1. Alors la matrice associée à cette numérotation est un sudoku associatif. Réciproquement, si P est un sudoku associatif d'ordre n, alors il définit une loi de groupe sur $\{1,...,n\}$ pour laquelle 1 est l'élément neutre (tout sudoku associatif vient d'un groupe).*

Ainsi, pour trouver tous les groupes d'ordre n (à isomorphisme près), il suffit de construire tous les sudokus d'ordre n, et ne garder que ceux qui sont associatifs. Sans oublier de ne garder qu'un exemplaire par classe d'isomorphisme. C'est ce que nous ferons avec Python.

Remarque 9. D'un point de vue strictement mathématique, la correspondance entre groupes finis et sudokus associatifs n'a pas grand intérêt. Chercher à classifier les groupes finis en cherchant les sudokus est une démarche grossière. De plus, cette méthode devient inefficace dès que l'ordre devient grand. Pour une recherche intelligente (et intéressante), on consultera [1] et [3], par exemple. Le but du présent article est modeste : écrire un programme simple en langage Python capable de trouver tous les groupes d'ordre au plus 6 (à isomorphisme près) en un temps très court.

Exercice 3. Vérifier qu'il n'existe qu'un sudokus d'ordre 3. Pourquoi ce sudoku est-il forcément associatif ? Combien y a t-il de sudokus d'ordre 4 ?

Exercice 4. Expliquer pourquoi le fait d'avoir restreint la définition des *sudokus* aux matrices dont la première ligne et première colonnes coïncident avec $\begin{pmatrix} 1 & ... & n \end{pmatrix}$ (définition 6) rend la proposition 5 inutile pour démontrer la proposition 8.

3 Python et permutations

Pour détecter les éventuels isomorphismes entre groupes, notre programme aura besoin de générer toutes les permutations d'un degré donné. Dans cette section, nous montrons plusieurs manières d'y parvenir. Bien sûr, la liste des méthodes proposées ici n'est pas exhaustive. Le lecteur pressé peut sauter cette section.

3.1 Convention

On rappelle qu'une permutation d'un ensemble E est une bijection de E sur lui-même. L'ensemble des permutations de E muni de la loi de composition des applications est un groupe noté $\mathcal{S}(E)$. Le groupe des permutations de $\{1,...,n\}$ est appelé groupe symétrique de degré n, et se note $\mathcal{S}_n$. Donnons un exemple. La permutation de $\{1,2,3\}$ définie par

$$\begin{array}{ccc} 1 & \longmapsto & 3 \\ 2 & \longmapsto & 1 \\ 3 & \longmapsto & 2 \end{array}$$

se note

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

(les antécédents en haut et les images en bas). En informatique, on encode cette bijection avec l'uplet $(3,1,2)$, tout simplement. Nous ferons de même.

Exercice 5. Lister toutes les permutations de $\{1, 2, 3\}$ (en suivant notre convention).

3.2 Algorithme par itérations

Nous proposons ici un programme bestial (et peu efficace) pour calculer l'ensemble des permutations de $\{1, 2, 3\}$. Il consiste à générer tous les éléments de $\{1, 2, 3\}^3$ et ne conserver que les uplets sans répétitions.

```
# permutations de {1,2,3}
L = {1,2,3}
PERM = []
for x in L:
    for y in L:
        for z in L:
            if len({x,y,z})==3:
                PERM = PERM + [(x,y,z)]
```

On voit mal comment convertir ce code en une fonction capable de calculer l'ensemble des permutations d'un itérable de longueur inconnue *a priori*. Néanmoins, l'idée exposée ici peut nous dépanner dans des situations particulières (quand il s'agit d'un besoin très simple).

Remarque 10. Les objets de type `set`, comme par exemple `{1,2,3}`, sont étudiés au chapitre 23 de [5]. Les objets de type `list`, comme `PERM`, au chapitre 20. Les objets de type `tuple`, comme `(x,y,z)`, au chapitre 21. La fonction `len` renvoie la longueur de n'importe quel objet itérable, de sorte que `len({x,y,z})` renvoie le cardinal de l'ensemble `{x,y,z}`. La ligne `PERM = PERM + [(x,y,z)]` peut être remplacée par `PERM.append((x,y,z))`.

3.3 Fonction `itertools.permutations`

Si on envoie un itérable `L` à la fonction `permutations` du module `itertools`, cette dernière retourne un *iterator* générant toutes les permutations de `L`. Importons `permutations` depuis le module `itertools` :

```
>>> from itertools import permutations
```

Créons un itérable nommé `L` (une liste par exemple) :

```
>>> L = [1,2,3]
```

Pour avoir accès aux éléments de `permutations(L)`, on peut par exemple itérer avec une boucle `for` :

```
>>> for s in permutations(L):
        print(s)

(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

On retrouve bien les 6 permutations de `[1,2,3]`. Chacune d'elles est encodée par un `tuple`.

Remarque 11. Les modules (et les imports) sont décrits au chapitre 5 de [5]. Les listes au chapitre 20. Les uplets (`tuple`) au chapitre 21. Les *itérators* (*generators*) et le module `itertools` au chapitre 22.

On peut donner à `permutations` une liste contenant des répétitions :

```
>>> for s in permutations([1,1,2]):
        print(s,end=" ")

(1, 1, 2) (1, 2, 1) (1, 1, 2) (1, 2, 1)
(2, 1, 1) (2, 1, 1)
```

Dans tous ces exemples, les items de l'itérable envoyé à `permutations` étaient des nombres entiers, mais ceci n'est nullement une obligation. On peut envoyer une liste de chaînes de caractères, ou une liste d'items de n'importe quelle nature.

3.4 Une fonction récursive

Dans cette section, nous écrivons nous-mêmes une fonction calculant les permutations. Son utilisation est simple : on lui donne un itérable, et elle retourne la liste des permutations de cet itérable. Chaque permutation est encodée par une liste.

```
def perm(L):
    if not L:
        return []
    elif len(L)==1:
        return [list(L)]
    else:
        return [[L[i]]+P for i in range(len(L)) for P in perm(L[:i]+L[i+1:])]
```

La première instruction équivaut à ceci : si `L` est vide, on retourne `[]` (liste vide). La deuxième instruction équivaut à ceci : si `L` ne possède qu'un item, on retourne `[list(L)]`, liste contenant l'unique permutation de `L` (la permutation triviale, encodée par une liste). La troisième instruction est récursive (voir plus bas).

Remarque 12. La structure `if...elif...else` est décrite au chapitre 9 de [5]. Les booléens sont décrits au chapitre 17 (on comprendra alors l'instruction `if not L`). La syntaxe pour écrire une fonction (`def`, `return`, etc.) est décrite au chapitre 13.

La récursivité écrite ici est basée sur une idée simple. Regardons un exemple : pour écrire toutes les permutations de $\{1, 2, 3\}$, on commence par les permutations $(1, *, *)$, ensuite on passe aux $(2, *, *)$, et on termine avec les $(3, *, *)$. Pour obtenir tous les $(1, *, *)$, on concatène (1) avec chaque permutation de $\{2, 3\}$. Pour obtenir les $(2, *, *)$, même chose : on concatène (2) avec chaque permutation de $\{1, 3\}$ et ainsi de suite. Tout le monde voit la relation de récursivité : chaque permutation de $(1, 2, 3)$ s'écrit en concaténant chaque (a) avec chaque permutation de $\{b, c\}$, ce dernier étant l'ensemble obtenu en retirant a de $\{1, 2, 3\}$. Cette concaténation s'écrit `[L[i]]+P` dans notre script, où `L[i]` décrit tous les items de `L`, et `P` décrit toutes les permutations de la liste « `L` privée de `L[i]` ». Pour que `L[i]` décrive tous les items de `L`, il suffit que `i` décrive toutes les positions possibles dans `L`, c'est ce que nous provoquons en écrivant `for i in range(len(L))`. Pour que `P` décrive toutes les permutations de la liste `L` privée de `L[i]`, nous avons écrit `for P in perm(L[:i]+L[i+1:])`. Le lecteur vérifiera que `L[:i]+L[i+1:]` est bien la liste `L` privée de son item `[i]`.

Remarque 13. Le type `range` est décrit au chapitre 16 de [5]. La fonction `len` retourne la longueur d'un itérable. Les syntaxes `L[:i]` et `L[i:]` sont décrites au chapitre 20 (consacré aux listes).

On observera que notre fonction retourne bien la liste de tous les `L[i]+P` attendus grâce à une écriture typique de Python : l'écriture en *compréhension*. Schématiquement, la fonction retourne :

```
[  [L[i]]+P  for i in here for P in there  ]
```

Remarque 14. L'écriture d'une liste en compréhension est décrite en détail aux sections 20.9 et 20.10 de [5] (chapitre 20 consacré aux listes).

La récursivité s'arrête forcément dans cette fonction. En effet, l'appel `perm(L)` déclenche des appels `perm(L[:i]+L[i+1:])` où la liste `L[:i]+L[i+1:]` est strictement plus courte que la liste initiale. On finit donc par tomber sur des listes de longueur 1, c'est-à-dire le cas traité dès la deuxième instruction (arrêt de la récursivité).

Exercice 6. Exécuter à la main la fonction `perm` pour `L=[1,2,3]`.

3.5 Un algorithme de Donald Knuth

On trouve dans [2] un algorithme basé sur la notion d'ordre lexicographique. L'idée consiste à munir chaque groupe $\mathcal{S}_n$ de cette relation d'ordre. En vertu de la convention faite à la section 3.1, l'ensemble $\mathcal{S}_3$, par exemple, s'écrit

$$\mathcal{S}_3 = \{(1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)\}$$

et comparer lexicographiquement $(2,1,3)$ avec $(3,1,2)$ revient à comparer les entiers 213 et 312. On trouve bien sûr $(2,1,3) < (3,1,2)$. Le lecteur attentif aura remarqué que nous avons listé les éléments de $\mathcal{S}_3$ dans l'ordre croissant. Le noyau dur de l'algorithme de Knuth est une fonction capable de calculer la permutation qui arrive *juste après* une permutation donnée, selon l'ordre lexicographique. Cette fonction, nous l'appellerons `suivant`, de sorte que `suivant([2,1,3])` retournera `[2,3,1]`. L'argument de cette fonction est obligatoirement une liste (ie. une donnée de type `list`). Voici la traduction en langage Python de la fonction `suivant` :

```
def suivant(L):
    K = L.copy()
    if len(K)<=1:
        return None
    i = len(K) - 2
    while K[i]>=K[i+1]:
        i = i - 1
    if i<0:
        return None
    j = len(K) - 1
    while K[j]<=K[i]:
        j = j - 1
    K[i],K[j] = K[j],K[i]
    i = i + 1
    j = len(K) - 1
    while j>=i:
        K[i],L[j] = K[j],K[i]
        i = i + 1
        j = j - 1
    return K
```

Remarque 15. La fonction `suivant` est intéressante du point de vue mathématique et algorithmique. Le lecteur pourra l'étudier dans [2]. Son code Python en revanche ne présente pas grand intérêt, si ce n'est la nécessité dès la première ligne de faire une copie de la liste `L` avec la méthode `copy`, afin d'éviter un *effet de bord*. Ces derniers sont décrits à la section 13.13 du chapitre 13 de [5]. La méthode `copy` et le phénomène d'*aliasing* sont largement documentés dans [5].

Testons la fonction `suivant` :

```
>>> suivant([2,3,1])
[3, 1, 2]
>>> suivant([3,1,2])
[3, 2, 1]
```

```
>>> suivant([3,2,1])
>>>
```

On peut même demander quelle est la « permutation » de $(1, 1, 1, 3)$ — on nous pardonnera cet abus de langage — qui vient juste après $(1, 1, 3, 1)$:

```
>>> suivant([1,1,3,1])
[1, 3, 1, 1]
```

Analysons ce que fait la fonction `suivant` pour construire `suivant([10,3,4,9,1,5,8,7,6,2])`. D'abord, elle cherche deux items consécutifs `a,b` tels que `a<b`, en commençant par la fin `6,2]`. Elle trouve `5,8` :

10	3	4	9	1	5	8	7	6	2

Ensuite elle cherche l'item strictement supérieur à notre `5` grisé sur la figure, en commençant par la fin, `2`. Elle trouve `6` :

10	3	4	9	1	5	8	7	6	2

En position `[i]` nous avons `5`, et en `[j]` nous avons `6`. La fonction permute ces deux items,

10	3	4	9	1	6	8	7	5	2

augmente `[i]` d'un cran, positionne `[j]` à la fin :

10	3	4	9	1	6	8	7	5	2

et entame une série de permutations selon le principe suivant : permutation des deux items gris, augmentation de `[i]`, diminution de `[j]`, permutation des deux gris, et ainsi de suite jusqu'à ce que les deux items gris se rejoignent (ou se croisent) :

10	3	4	9	1	6	2	7	5	8

10	3	4	9	1	6	2	7	5	8

10	3	4	9	1	6	2	5	7	8

La fonction retourne $(10, 3, 4, 9, 1, 6, 2, 5, 7, 8)$.

Maintenant que nous avons compris à quoi sert la fonction `suivant`, nous pouvons écrire un algorithme utilisant cette dernière pour construire la liste des permutations d'une liste donnée :

```
def knuth(L):
    if len(L)<=1:
        return [L]
    resultat = []
    while L:
        resultat = resultat + [L]
        L = suivant(L)
    return resultat
```

Si les items de `L` sont dans l'ordre croissant, l'appel `knuth(L)` retourne la liste des permutations de `L` :

```
>>> knuth([1,2,3])
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Si les items de `L` ne sont pas dans l'ordre croissant, on se doute du résultat :

```
>>> knuth([2,3,1])
```

```
[[2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Exercice 7. Si `L` contient la liste vide `[]`, les appels `perm(L)` (voir section 3.4) et `knuth(L)` ne retournent pas la même chose. D'un point de vue mathématique, laquelle a raison ?

Exercice 8. Réécrire la fonction `knuth` afin que `knuth(L)` retourne la liste des permutations de `L` même si cette dernière n'est pas donnée dans l'ordre croissant. Indication : utiliser la fonction `sorted` ou la méthode `sort`.

On peut modifier `knuth` pour en faire un *generator*, à l'instar de `itertools.permutations` :

```
def knuth(L):
    if len(L)<=1:
        yield L
        raise StopIteration
    while 1:
        yield L
        i = len(L) - 2
        while L[i]>=L[i+1]:
            i = i - 1
        if i<0:
            raise StopIteration
        j = len(L) - 1
        while L[j]<=L[i]:
            j = j - 1
        L[i],L[j] = L[j],L[i]
        i = i + 1
        j = len(L) - 1
        while j>=i:
            L[i],L[j] = L[j],L[i]
            i = i + 1
            j = j - 1
```

Dans ce cas, `knuth` retourne non pas une liste mais un objet assez complexe (un *generator iterator*) qui a vocation à être itéré avec une boucle par exemple :

```
>>> for s in knuth([1,2,3]):
        print(s,end=" ")

[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1]
[3, 1, 2] [3, 2, 1]
```

Remarque 16. La dernière version de `knuth` exploite des spécificités fortes de Python telles que `yield` et `raise`. L'instruction `yield` est étudiée au chapitre 22 de [5] (chapitre consacré aux générateurs), et l'instruction `raise` au chapitre 12 (gestion des exceptions avec `try-except-else`).

3.6 Algorithme par insertions

Nous nous contenterons de décrire l'algorithme par insertions sur un exemple simple : le calcul de l'ensemble des permutations de $\{1,2,3,4\}$. Pour ce faire, l'algorithme commence par écrire les permutations de $\{1\}$, puis celles de $\{1, 2\}$, puis celles de $\{1, 2, 3\}$, pour enfin produire celles de $\{1,2,3,4\}$.

- Permutations de $\{1\}$: (1).
- Permutations de $\{1, 2\}$: on insère 2 dans chaque permutation de $\{1\}$. On trouve : $(2, 1)$, $(1,2)$.
- Permutations de $\{1,2,3\}$: on insère 3 dans chaque permutation de $\{1,2\}$:
  - $(2,1)$ donne $(3,2,1)$, $(2,3,1)$, $(2,1,3)$.
  - $(1,2)$ donne $(3,1,2)$, $(1,3,2)$, $(1,2,3)$.

- Permutations de $\{1,2,3,4\}$: on insère 4 dans chaque permutation de $\{1,2,3\}$.
  - $(3,2,1)$ donne $(4,3,2,1)$, $(3,4,2,1)$, $(3,2,4,1)$, $(3,2,1,4)$.
  - etc.

Exercice 9. Traduire l'algorithme par insertions en langage Python.

4 Programme pour une recherche exhaustive des groupes

Dans cette section, une matrice d'ordre n est une matrice carrée $n\times n$ à coefficients dans l'ensemble $\{0,\ldots,n-1\}$. Nous aurions préféré travailler avec des coefficients dans $\{1,\ldots,n\}$, mais nous avons préféré nous adapter à Python qui, comme chacun sait, numérote les items d'une liste en partant de 0 et non de 1. Nous encodons une matrice d'ordre n avec une liste de listes. Par exemple, la matrice

$$M=\begin{pmatrix}1&1&2\\1&1&0\\2&1&2\end{pmatrix}$$

est encodée par :

```
>>> M = [[1,1,2],[1,1,0],[2,1,2]]
```

Le coefficient grisé s'appelle normalement M_{23} (ligne 2, colonne 3), mais nous préférons nous adapter à Python en l'appelant M_{12} :

```
>>> M[1][2]
0
```

Nous regarderons M comme la table de Pythagore d'une loi interne de $\{0,\ 1,\ 2\}$ notée $\times$. Par exemple, ici nous avons :

$$1\times 2=0$$

4.1 Fonction générant tous les sudokus d'ordre n

La fonction `sudoku` présentée ici reçoit un entier naturel non nul n en argument, et retourne la liste de tous les sudokus d'ordre n. Montrons sur l'exemple `n=4` comment cette fonction travaille. En fait, elle fabrique un arbre généalogique. La première génération est $\begin{pmatrix}0&1&2&3\end{pmatrix}$, ancêtre commun de tous les sudokus d'ordre 4. Pour construire les enfants de ce dernier, on calcule les lignes acceptables (celles qui conservent la propriété d'être un sudoku, nous montrerons plus loin comment). On en trouve trois : $\begin{pmatrix}1&0&3&2\end{pmatrix}$, $\begin{pmatrix}1&2&3&0\end{pmatrix}$ et $\begin{pmatrix}1&3&0&2\end{pmatrix}$. On peut donc dire que $\begin{pmatrix}0&1&2&3\end{pmatrix}$ enfante $\begin{pmatrix}0&1&2&3\\1&0&3&2\end{pmatrix}$, $\begin{pmatrix}0&1&2&3\\1&2&3&0\end{pmatrix}$ et $\begin{pmatrix}0&1&2&3\\1&3&0&2\end{pmatrix}$. On recommence avec chacun de ces trois presque-sudokus. Voici le résultat final :

$$\begin{pmatrix}0&1&2&3\end{pmatrix}$$

$$\begin{pmatrix}0&1&2&3\\1&0&3&2\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&2&3&0\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&3&0&2\end{pmatrix}$$

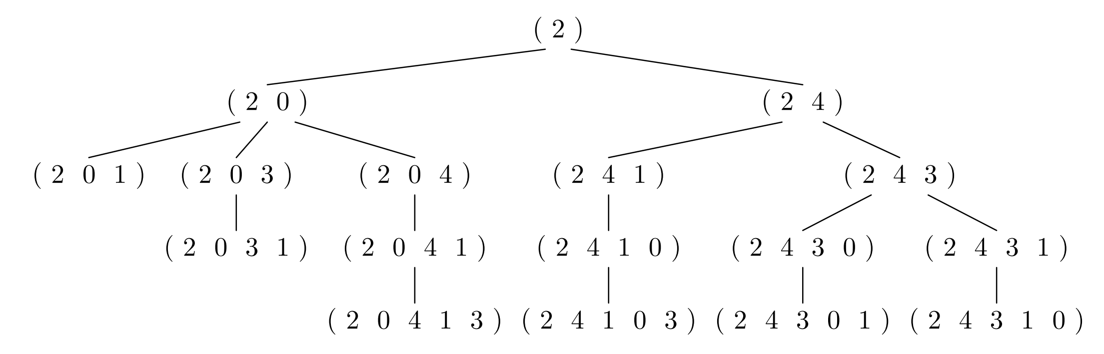
$$\begin{pmatrix}0&1&2&3\\1&0&3&2\\2&3&0&1\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&0&3&2\\2&3&1&0\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&2&3&0\\2&3&0&1\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&3&0&2\\2&0&3&1\end{pmatrix}$$

$$\begin{pmatrix}0&1&2&3\\1&0&3&2\\2&3&0&1\\3&2&1&0\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&0&3&2\\2&3&1&0\\3&2&0&1\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&2&3&0\\2&3&0&1\\3&0&1&2\end{pmatrix}\quad\begin{pmatrix}0&1&2&3\\1&3&0&2\\2&0&3&1\\3&2&1&0\end{pmatrix}$$

Nous voyons que l'appel `sudoku(4)` retournera quatre sudokus d'ordre 4. Montrons maintenant comment on calcule les lignes qu'un presque-sudoku peut accepter. Prenons par exemple

$$A = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \end{pmatrix}$$

Il s'agit bien d'un presque-sudoku. Ici aussi, la méthode consiste à fabriquer un arbre généalogique. La première génération est la presque-ligne $(\,2\,)$. Passons à la deuxième génération. Après le 2, on ne peut pas mettre un 2. De plus, la deuxième colonne de A contient déjà 1 et 3, donc les seuls coefficients acceptables sont 0 et 4. Autrement dit, les enfants de $(\,2\,)$ sont $(\,2\ \ 0\,)$ et $(\,2\ \ 4\,)$; c'est la deuxième génération. On passe à la troisième génération. D'après le même raisonnement, $(\,2\ \ 0\,)$ donne $(\,2\ \ 0\ \ 1\,)$, $(\,2\ \ 0\ \ 3\,)$ et $(\,2\ \ 0\ \ 4\,)$, tandis que $(\,2\ \ 4\,)$ donne $(\,2\ \ 4\ \ 1\,)$ et $(\,2\ \ 4\ \ 3\,)$. On continue ainsi de suite jusqu'à la cinquième génération :



Ainsi, les lignes que A peut accepter sont $(\,2\ \ 0\ \ 4\ \ 1\ \ 3\,)$, $(\,2\ \ 4\ \ 1\ \ 0\ \ 3\,)$, $(\,2\ \ 4\ \ 3\ \ 0\ \ 1\,)$ et $(\,2\ \ 4\ \ 3\ \ 1\ \ 0\,)$. On en déduit les enfants de A :

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \\ 2 & 0 & 4 & 1 & 3 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \\ 2 & 4 & 1 & 0 & 3 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \\ 2 & 4 & 3 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 3 & 0 & 4 & 2 \\ 2 & 4 & 3 & 1 & 0 \end{pmatrix}$$

Voici le code de notre fonction `sudoku` :

```
def sudoku(n):
    GC = [[list(range(n))]]
    for i in range(1,n):
        GN = []
        for M in GC:
            DM = []  # contiendra les P-S enfantés par M
            gc = [[i]]
            for j in range(1,n):
                gn = []
                for L in gc:
                    DL = [] # contiendra les P-L enfantées par L
                    support = set(range(n)) - set(L)
                    support = support - {M[i][j] for i in range(0,i)}
                    for k in support:
                        DL = DL + [L+[k]]  # L+[k] est un enfant de L
                    gn = gn + DL
                gc = gn
            for L in gc:
                DM = DM + [M+[L]]   # M+[L] est un enfant de M
            GN = GN + DM
        GC = GN
```

```
    return GC
```

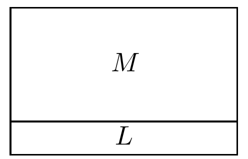
Variables utilisées :

- GC : génération courante (contient les presque-sudokus de la génération en cours).
- GN : génération nouvelle. C'est la génération venant juste après GC. Chaque presque-sudoku M de GC enfante une liste DM (c'est la liste des presque-sudokus dérivant de M). Ainsi, pour chaque M de GC, on enrichit GN avec les éléments de DM grâce à l'instruction GN = GN + DM.
- gc : génération courante pour les lignes (les presque-lignes de la génération en cours).
- gn : génération nouvelle. C'est la génération venant juste après gc. Chaque presque ligne L de gc enfante une liste DL (c'est la liste des presque-lignes dérivant de L). Ici aussi nous retrouvons naturellement l'instruction gn = gn + DL.

Remarque 17. Les listes et la fonction list font l'objet du chapitre 20 de [5]. Les boucles for font l'objet du chapitre 10. Les ensembles et la fonction set font l'objet du chapitre 23. La définition d'un ensemble en compréhension comme par exemple {M[i][j] for i in range(0,i)} y est expliquée en détail.

4.2 Alternative bestiale pour sudoku

Au lieu de calculer intelligemment l'ensemble des lignes acceptables pour un presque-sudoku donné, on peut tout simplement générer toutes les lignes possibles, c'est-à-dire toutes les permutations de $(\,0\ \ 1\ \ \ldots\ \ n-1\,)$, et les tester une par une pour savoir lesquelles on garde. Soit M un presque-sudoku de dimensions $m \times n$ $(m < n)$. Soit L une ligne de longueur n (c'est-à-dire une matrice $1 \times n$). On écrit ici une fonction retournant True si l'ajout de L à M conserve la propriété d'être un sudoku, False sinon :



Il est clair que L est acceptable si et seulement si pour tout j, le coefficient L_j ne se trouve pas dans la colonne j de M. Voici la fonction :

```
def accepte(M,L):
    n = len(M[0])
    m = len(M)
    for j in range(n):
        if L[j] in [M[i][j] for i in range(m)]:
            return False
    return True
```

La fonction bestiale remplaçant sudoku serait alors :

```
def sudoku_bestial(n):
    GC = [[list(range(n))]]
    for i in range(1,n):
        support = list(range(n))
        support.remove(i)  # liste de 0 à n-1 sauf i
        GN = []
        for M in GC:
            DM = []
            for ligne in permutations(support):
                ligne = [i] + list(ligne)
                if accepte(M,ligne):
```

```
                    DM = DM + [M+[ligne]]
            GN = GN + DM
        GC = GN
    return GC
```

On notera que `sudoku_bestial` utilise la fonction `permutations` du module `itertools`.

Remarque 18. Les méthodes ordinaires appartenant au type `list`, comme par exemple la méthode `remove` appliquée à la liste nommée `support` dans ce code (ligne 6), sont décrites à la section 20.13 de [5].

4.3 Fonction pour savoir si une matrice est associative

Dire qu'une matrice M d'ordre n est associative, signifie que la loi $\times$ qu'elle définit sur $\{0, \ldots, n-1\}$ vérifie la propriété

$$(i\,j)\,k = i\,(j\,k)$$

autrement dit que

$$M_{M_{ij},k} = M_{i,M_{jk}}$$

Voici le code de notre fonction testant l'associativité :

```
def associative(M):
    n = len(M)
    for a in range(n):
        for b in range(n):
            for c in range(n):
                if M[M[a][b]][c]!=M[a][M[b][c]]:
                    return False
    return True
```

Une fois construits tous les sudokus d'ordre n, nous les testerons un par un pour ne garder que les associatifs. On rappelle que chaque sudoku associatif représente un groupe.

Remarque 19. Les opérateurs de comparaison comme `!=` (qui signifie $\neq$) sont étudiés à la section 17.2 du chapitre 17 (booléens) de [5].

4.4 Fonction pour savoir si une application est un isomorphisme de groupes

Soient M et N des sudokus associatifs d'ordre n. Chacune de ces matrices est considérée comme la table de Pythagore d'une loi de groupe sur $\{0, \ldots, n-1\}$. On note $\mathcal{M}$ et $\mathcal{N}$ les groupes définis par M et N respectivement. Soit f une bijection de $\{0, \ldots, n-1\}$. On écrit ici une fonction `isomorphisme` répondant `True` si $f: \mathcal{M} \to \mathcal{N}$ est un isomorphisme de groupes, et `False` sinon. Il s'agit de vérifier que l'on a

$$f(a\,b) = f(a)\,f(b)$$

pour tous $a, b \in \{0, \ldots, n-1\}$. Autrement dit, vérifier que l'on a

$$f(M_{ab}) = N_{f(a),f(b)}$$

Dans notre programme, la fonction `isomorphisme` ne traitera que des bijections f vérifiant $f(0) = 0$. On rappelle que les premières ligne et colonne d'un sudoku sont par définition triviales, de sorte que tout sudoku associatif est la table d'une loi de groupe sur $\{0, \ldots, n-1\}$ où 0 joue le rôle d'élément neutre. Il est donc inutile de tester l'égalité $f(a\,b) = f(a)\,f(b)$ lorsque a ou b est nul.

```
def isomorphisme(f,M,N):
    n = len(M)
```

```
    for a in range(1,n):
        for b in range(1,n):
            if f[M[a][b]]!=N[f[a]][f[b]]:
                return False
    return True
```

Remarque 20. D'un point de vue mathématique, `f` est une bijection, mais d'un point de vue informatique, `f` est un `tuple` (nous rappelons que chaque permutation est encodée ici avec un uplet), c'est la raison pour laquelle dans notre script, une image $f(x)$ s'écrit `f[x]`, et non pas `f(x)`.

4.5 Fonction pour savoir si deux groupes sont isomorphes

Nous reprenons les notations de la sous-section précédente. Nous proposons un algorithme bestial pour savoir si $\mathcal{M}$ et $\mathcal{N}$ sont isomorphes. La fonction `isomorphe` regarde s'il existe une bijection $f\colon\mathcal{M}\to\mathcal{N}$ fixant 0 qui soit une morphisme de groupe.

```
def isomorphe(M,N):
    n = len(M)
    for f in permutations(range(1,n)):
        f = (0,) + f             # f(0)=0
        if isomorphisme(f,M,N):
            return f
    return False
```

La fonction `permutations` appelée à la troisième ligne est celle qui est offerte par le module `itertools` (section 3.3). Il faudra penser à l'importer.

Remarque 21. Les objets de type `tuple`, comme le `f` et le `(0,)` de ce code sont étudiés au chapitre 21 de [5]. D'un point de vue mathématique, un objet de type `tuple` est un uplet. L'instruction `return f` (ligne 6) aura le même effet qu'un `return True`. Pour comprendre pourquoi, on pourra consulter le chapitre 17 de [5] consacré aux booléens.

4.6 Fonction pour savoir si un groupe donné est déjà représenté dans la liste

Chaque fois que nous générons un groupe, nous regardons si notre stock ne contient pas déjà un groupe qui lui serait isomorphe. La fonction `est_dans` reçoit deux arguments : un sudoku associatif `M` et une liste de sudokus associatifs `L`.

```
def est_dans(M,L):
    for K in L:
        if isomorphe(M,K):
            return True
    return False
```

4.7 Fonction générant tous les groupes à isomorphisme près

La fonction `groupes` reçoit un entier naturel `n` non nul en argument, et retourne la liste de tous les groupes d'ordre `n` à isomorphisme près. Ainsi, chaque classe d'isomorphisme possède un représentant et un seul dans la liste retournée.

```
def groupes(n):
    S = sudoku(n)   # tous les sudokus d'ordre n
    G = []
    for M in S:
```

```
        if associative(M) and not est_dans(M,G):
            G = G + [M]
    return G
```

Remarque 22. Quand Python calcule le booléen `associative(M) and not est_dans(M,G)` lors de l'instruction `if` de la ligne 5, il commence par calculer `associative(M)`. Si celui-ci vaut `False`, il arrête le calcul car il sait que le booléen cherché vaut `False` lui aussi. Si tel n'avait été le cas, nous aurions décomposé ce simple `if` en un double `if` afin de gagner en vitesse d'exécution. Nous décrivons la manière dont Python utilise les opérateurs logiques tels que `and` et `or` aux sections 17.5 et 17.6 de [5].

5 Résultats de notre recherche

Nous ne parlerons pas du cas `n = 1`. Nous avons utilisé le *décorateur* décrit à la section 14.7.3 du livre [5] pour chronométrer notre programme.

5.1 Groupes d'ordre 2

- Recherche des sudokus : 0,000 02 secondes (1 seul sudoku).
- Recherche des groupes : 0,000 03 secondes (1 seul groupe).

Groupe trouvé : $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. On reconnaît $\mathbb{Z}/2\,\mathbb{Z}$. C'est conforme à la théorie, Cf [6].

Remarque 23. Nous avons chronométré le temps utilisé pour trouver les sudokus, puis le temps utilisé pour trouver les groupes parmi ces derniers.

5.2 Groupes d'ordre 3

- Recherche des sudokus : 0,000 05 secondes (1 seul sudoku).
- Recherche des groupes : 0,000 03 secondes (1 seul groupe).

Groupe trouvé : $\begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{pmatrix}$

On reconnaît le groupe $\mathbb{Z}/3\,\mathbb{Z}$. C'est conforme à la théorie.

5.3 Groupes d'ordre 4

- Recherche des sudokus : 0,000 1 secondes (4 sudokus, tous associatifs).
- Recherche des groupes : 0,000 1 secondes (2 groupes).

Groupes trouvés : $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$ et $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 1 & 0 \\ 3 & 2 & 0 & 1 \end{pmatrix}$

On reconnaît les groupes $(\mathbb{Z}/2\,\mathbb{Z})^2$ et $\mathbb{Z}/4\,\mathbb{Z}$, respectivement. C'est conforme à la théorie. Voici une activité permettant de comprendre comment on a fait pour reconnaître $\mathbb{Z}/4\,\mathbb{Z}$. On note $\mathbb{Z}/4\,\mathbb{Z} = \{0, 1, 2, 3\}$. La table de ce groupe est alors

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

On compare les deux tables avec la fonction `isomorphe` :

```
>>> G2 = [[0, 1, 2, 3],[1, 0, 3, 2],[2, 3, 1, 0],[3, 2, 0, 1]]
```

```
>>> Z4 = [[0, 1, 2, 3],[1, 2, 3, 0],[2, 3, 0, 1],[3, 0, 1, 2]]
>>> isomorphe(Z4,G2)
(0, 2, 1, 3)
```

Python nous répond en donnant un isomorphisme.

5.4 Groupes d'ordre 5

– Recherche des sudokus : 0,002 secondes (56 sudokus, parmi eux 6 associatifs).
– Recherche des groupes : 0,000 7 secondes (1 groupe).

Groupe trouvé : $\begin{pmatrix} 0&1&2&3&4\\ 1&2&3&4&0\\ 2&3&4&0&1\\ 3&4&0&1&2\\ 4&0&1&2&3 \end{pmatrix}$

On reconnaît le groupe $\mathbb{Z}\,/\,5\,\mathbb{Z}$. C'est conforme à la théorie. Voici une activité permettant de comprendre comment on a fait pour savoir que parmi les 56 sudokus trouvés, il y en a 6 associatifs :

```
>>> S = sudoku(5)
>>> A = [M for M in S if associative(M)]
>>> len(A)
6
```

Note. Le lecteur a compris que les 6 sudokus associatifs trouvés représentent le « même » groupe.

5.5 Groupes d'ordre 6

– Recherche des sudokus : 0,75 secondes (9408 sudokus, parmi eux 80 associatifs).
– Recherche des groupes : 0,12 secondes (2 groupes).

Groupes trouvés : $\begin{pmatrix} 0&1&2&3&4&5\\ 1&0&3&2&5&4\\ 2&3&4&5&0&1\\ 3&2&5&4&1&0\\ 4&5&0&1&2&3\\ 5&4&1&0&3&2 \end{pmatrix}$ et $\begin{pmatrix} 0&1&2&3&4&5\\ 1&0&3&2&5&4\\ 2&4&0&5&1&3\\ 3&5&1&4&0&2\\ 4&2&5&0&3&1\\ 5&3&4&1&2&0 \end{pmatrix}$

Le lecteur vérifiera que le premier est $\mathbb{Z}\,/\,6\,\mathbb{Z}$ et le deuxième est $\mathcal{S}_3$ (groupe symétrique de degré 3).

5.6 Groupes d'ordre 7

Nous n'avons pas eu la patience d'attendre la fin de l'exécution. La fonction `sudoku` nécessite 9 minutes pour construire la quatrième génération de presque-sudokus. Les chiffres sont spectaculaires :

– deuxième génération : 309 presque-sudokus ;
– troisième génération : 35 792 presque-sudokus ;
– quatrième génération : 1 293 216 presque-sudokus.

6 Programme pour une pêche aléatoire

Pour une pêche aux groupes basée sur la chance, nous aurons besoin d'une fonction générant aléatoirement une permutation :

```
def permutation_alea(n):
    S = list(range(n))     # S reçoit [0,...,n-1]
    shuffle(S)             # mélange les items de S
    return S
```

Attention : cette fonction utilise la fonction `shuffle` du module `random`. On n'oubliera pas de l'importer au début du script :

```
from random import shuffle
```

Remarque 24. Le module `random` est étudié en détail à la section 7.11 du chapitre 7 (calcul et arithmétique) du livre [5].

Nous aurons ensuite besoin d'une fonction générant aléatoirement un sudoku :

```
def sudoku_alea(n):
    M = [list(range(n))]
    for i in range(1,n):
        continuer = True
        while continuer:
            L = permutation_alea(n)
            L.remove(i)
            ligne = [i] + L
            if accepte(M,ligne):
                continuer = False
        M = M + [ligne]
    return M
```

Nous aurons enfin besoin d'une fonction générant successivement des sudokus jusqu'à tomber (par hasard) sur un sudoku associatif (ie un groupe). Cette fonction se comporte comme un pêcheur sélectif.

```
def groupe_alea(n):
    while 1:
        M = sudoku_alea(n)
        if associative(M):
            return M
```

Remarque 25. L'instruction `while 1` équivaut à `while True` (boucle infinie). Le chapitre 17 de [5] explique comment Python transforme automatiquement n'importe quel type de donnée (en l'occurence ici l'entier `1`) en un booléen.

Nous proposons d'organiser la pêche comme ceci :

```
# Programme principal (pêche)
S = [[]]*7
for n in range(2,7):
    for i in range(30):
        M = groupe_alea(n)
        if M not in S[n] and not est_dans(M,S[n]):
            S[n] = S[n] + [M]
```

Quelques explications : à la fin de l'exécution, `S[n]` contient tous les groupes d'ordre `n` pêchés (maximum un groupe par classe d'isomorphisme). Le programme ne traite que les cas $n \in \{2, 3, 4, 5, 6\}$. Pour chaque n, on appelle 30 fois la fonction `groupe_alea`. À chaque appel, on regarde si le groupe obtenu est nouveau ou pas. S'il est nouveau, on le stocke, sinon, on l'abandonne.

Voici le produit de notre pêche (juste une exécution) :

– `S[2]` a retenu $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, c'est-à-dire $\mathbb{Z}/2\,\mathbb{Z}$;

- `S[3]` a retenu $\begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{pmatrix}$, c'est-à-dire $\mathbb{Z}/3\,\mathbb{Z}$;
- `S[4]` a retenu $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \\ 2 & 0 & 3 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$ et $\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$, c'est-à-dire $\mathbb{Z}/4\,\mathbb{Z}$ et $(\mathbb{Z}/2\,\mathbb{Z})^2$;
- `S[5]` a retenu $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 0 \\ 2 & 3 & 4 & 0 & 1 \\ 3 & 4 & 0 & 1 & 2 \\ 4 & 0 & 1 & 2 & 3 \end{pmatrix}$, c'est-à-dire $\mathbb{Z}/5\,\mathbb{Z}$;
- `S[6]` a retenu $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 4 & 2 & 5 & 0 \\ 2 & 4 & 0 & 5 & 1 & 3 \\ 3 & 2 & 5 & 4 & 0 & 1 \\ 4 & 5 & 1 & 0 & 3 & 2 \\ 5 & 0 & 3 & 1 & 2 & 4 \end{pmatrix}$ et $\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 5 & 4 & 3 & 2 \\ 2 & 3 & 0 & 1 & 5 & 4 \\ 3 & 2 & 4 & 5 & 1 & 0 \\ 4 & 5 & 3 & 2 & 0 & 1 \\ 5 & 4 & 1 & 0 & 2 & 3 \end{pmatrix}$, c'est-à-dire $\mathbb{Z}/6\,\mathbb{Z}$ et $\mathcal{S}_3$.

Le hasard a voulu que nous attrapions tout ce qui existe ! Voici les durées enregistrées lors de cette exécution (arrondies au millionnième de seconde) :

`n`	durée en secondes
2	0,000 332
3	0,000 942
4	0,004 602
5	0,102 600
6	6,554 921

Il est intéressant de noter combien de fois la fonction `groupe_alea` doit appeler `sudoku_alea` pour tomber sur une loi associative (un groupe ici). Une petite modification de notre programme nous a permis d'obtenir les chiffres ci-dessous (sur une exécution) :

`n`	Nombre d'appels en moyenne
2	1
3	1
4	1
5	$\approx 6,6$
6	$\approx 115,7$

On constate que pour obtenir un sudoku associatif d'ordre 6, il faut, en moyenne, générer 116 sudokus. On rappelle que pour une exécution de notre programme de pêche, l'appel `groupe_alea(6)` a lieu 30 fois, c'est donc une moyenne sur 30 mesures.

7 Perspectives

Il y a énormément de choses à améliorer dans nos scripts. On peut les rendre plus rapides afin de pouvoir espérer traiter quelques cas au-delà du cas $n = 6$ (quelques jours après avoir rédigé cet article, l'auteur a résolu le cas $n = 7$). On notera par ailleurs que notre fonction `sudoku_alea` est incapable de trouver un sudoku d'ordre 13 en un temps raisonnable. Nous ne savons même pas si elle en est capable tout court ! Une activité intéressante, par exemple, consisterait à écrire une fonction permettant à coup sûr de générer un sudoku d'ordre 13 (ou plus) en s'inspirant d'un algorithme de type *backtracking search*, par exemple. Il y a beaucoup à faire...

8 Remerciements

L'auteur tient à remercier

- Jacques Legout (`http://tuogel.free.fr`) pour sa relecture attentive et les erreurs qu'il a eu la gentillesse de signaler.

– Benjamin Clerc pour sa relecture attentive, les erreurs qu'il a signalées, ses remarques constructives et la mise en ligne de cet article sur Mathématice.

– Dany-Jack Mercier pour son attention et la mise en ligne de cet article sur Mégamaths.

9 Bibliographie

[1] Daniel Perrin, *Cours d'algèbre*, Ellipses, 1996.

[2] Donald Knuth, *The art of computer programming*, Addison-Wesley, 1997.

[3] Pascal Ortiz, *Exercices d'algèbre*, Ellipses, 2004.

[4] Marc Reversat et Bruno Bigonnet, *Algèbre pour la licence*, Dunod, 2000.

[5] Richard Gomez, *Le petit Python, aide-mémoire pour Python 3*, Ellipses, 2017.

[6] Wikipedia, *Listes des petits groupes*, `https://fr.wikipedia.org/wiki/Liste_des_petits_groupes`.

[7] Jean-Pierre Petit et Jean-Marie Souriau, *L'axiome du sandwich*, `https://www.jp-petit.org/science/maths_f/ax_group_f/axiogr_f.htm`

[8] Revue Mathématice, `http://revue.sesamath.net`.

[9] Site Mégamaths, `http://megamaths.1free-host.free.com`.